課題S1解説
C言語編

中島 研吾
東京大学情報基盤センター

課題S1

- 内容
  - 「<$O-S1>/a1.0～a1.3」,「 <$O-S1>/a2.0～a2.3 」から局所ベクトル情報を読み込み，全体ベクトルのノルム（||x||）を求めるプログラムを作成する（S1-1）。
    - <$O-S1>file.f，<$O-S1>file2.fをそれぞれ参考にする。
  - 下記の数値積分の結果を台形公式によって求めるプログラムを作成する。MPI_reduce，MPI_Bcast等を使用して並列化を実施し，プロセッサ数を変化させた場合の計算時間を測定する（S1-3）。

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

ファイルコピー

FORTRANユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ cp /work/gt00/z30088/class_eps/F/s1r-f.tar .
>$ tar xvf s1r-f.tar
```

Cユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ cp /work/gt00/z30088/class_eps/C/s1r-c.tar .
>$ tar xvf s1r-c.tar
```

ディレクトリ確認

```
>$ ls
    mpi
>$ cd mpi/S1-ref
```

このディレクトリを本講義では **<$O-S1r>** と呼ぶ。

<$O-S1r> = <$O-TOP>/mpi/S1-ref

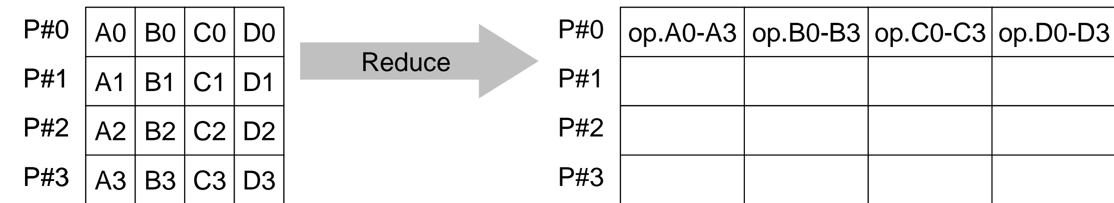
S1-1：局所ベクトル読み込み，ノルム計算

- 「<$O-S1>/a1.0～a1.3」，「 <$O-S1>/a2.0～a2.3 」から局所ベクトル情報を読み込み，全体ベクトルのノルム（||x||）を求めるプログラムを作成する（S1-1）。

- MPI_Allreduce（またはMPI_Reduce）の利用

- ワンポイントアドバイス
  - 変数の中身を逐一確認しよう！

MPI_Reduce



- コミュニケーター「comm」内の，各プロセスの送信バッファ「sendbuf」について，演算「op」を実施し，その結果を1つの受信プロセス「root」の受信バッファ「recbuf」に格納する。
  - 総和，積，最大，最小 他

- **MPI_Reduce (sendbuf,recvbuf,count,datatype,op,root,comm)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
  - **op** 整数 I 計算の種類
    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    ユーザーによる定義も可能： MPI_OP_CREATE
  - **root** 整数 I 受信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する

送信バッファと受信バッファ

- MPIでは「送信バッファ」,「受信バッファ」という変数がしばしば登場する。
- 送信バッファと受信バッファは必ずしも異なった名称の配列である必要はないが，必ずアドレスが異なっていなければならない。

MPI_Reduce/Allreduceの “op”

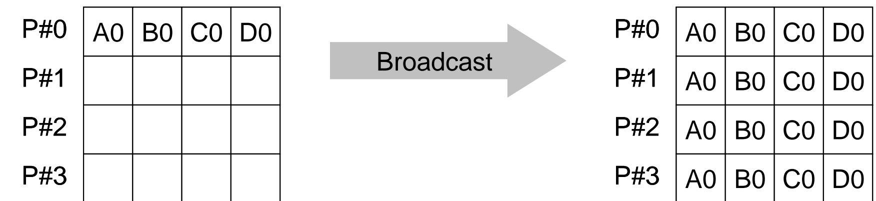
```
MPI_Reduce
 (sendbuf,recvbuf,count,datatype,op,root,comm)
```

- **MPI_MAX**, **MPI_MIN** 最大値，最小値
- **MPI_SUM**, **MPI_PROD** 総和，積
- **MPI_LAND** 論理AND

```
double X0, XSUM;

MPI_Reduce
(&X0, &XSUM, 1, MPI_DOUBLE, MPI_SUM, 0, <comm>)
```

```
double X0[4];

MPI_Reduce
(&X0[0], &X0[2], 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>)
```

MPI_Bcast



- コミュニケーター 「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から，その他全てのプロセスのバッファ「buffer」にメッセージを送信。

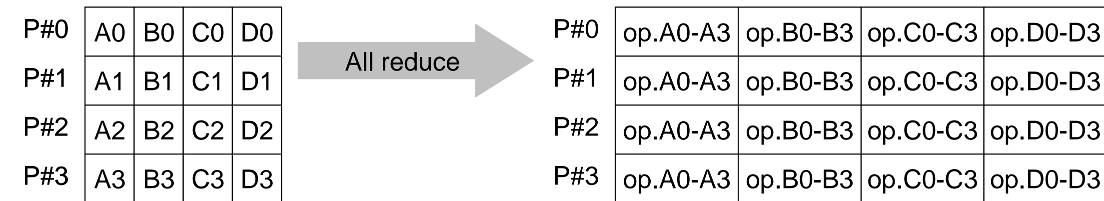
- **MPI_Bcast (buffer,count,datatype,root,comm)**
  - **buffer** 任意 I/O バッファの先頭アドレス，タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ

    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.

    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
  - **root** 整数 I 送信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する

S1-1

MPI_Allreduce



- MPI_Reduce + MPI_Bcast
- 総和，最大値を計算したら，各プロセスで利用したい場合が多い

- **call MPI_Allreduce (sendbuf,recvbuf,count,datatype,op, comm)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **op** 整数 I 計算の種類
  - **comm** 整数 I コミュニケータを指定する

S1-1：局所ベクトル読み込み，ノルム計算
均一長さベクトルの場合(a1.*)：s1-1-for_a1.c

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
        int i, N;
        int PeTot, MyRank;
        MPI_Comm SolverComm;
        double vec[8];
        double sum0, sum;
        char filename[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(filename, "a1.%d", MyRank);
        fp = fopen(filename, "r");
        assert(fp != NULL);

        N=8;
        for(i=0;i<N;i++){
                fscanf(fp, "%lf", &vec[i]);}
        sum0 = 0.0;
        for(i=0;i<N;i++){
                sum0 += vec[i] * vec[i];}

        MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        sum = sqrt(sum);

        if(!MyRank) printf("%27.20E¥n", sum);
        MPI_Finalize();
        return 0;
}
```

S1-1：局所ベクトル読み込み，ノルム計算

不均一長さベクトルの場合(a2.*)：s1-1-for_a2.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
        int i, PeTot, MyRank, n;
        MPI_Comm SolverComm;
        double *vec, *vec2;
        int * Count, CountIndex;
        double sum0, sum;
        char filename[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(filename, "a2.%d", MyRank);
        fp = fopen(filename, "r");
        assert(fp != NULL);

        fscanf(fp, "%d", &n);
        vec = malloc(n * sizeof(double));
        for(i=0;i<n;i++){
                fscanf(fp, "%lf", &vec[i]);}
        sum0 = 0.0;
        for(i=0;i<n;i++){
                sum0 += vec[i] * vec[i];}

        MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        sum = sqrt(sum);

        if(!MyRank) printf("%27.20E¥n", sum);
        MPI_Finalize();
        return 0;}
```

実 行(課題S1-1)

FORTRAN

```
$ cd /work/gt00/t00XXX/pFEM/mpi/S1-ref
$ mpiifort -O3 s1-1-for_a1.f
$ mpiifort -O3 s1-1-for_a2.f

(modify "go4.sh")
$ pjsub go4.sh
```

C

```
$ cd /work/gt00/t00XXX/pFEM/mpi/S1-ref
$ mpicc -O3 s1-1-for_a1.c
$ mpicc -O3 s1-1-for_a2.c

(modify "go4.sh")
$ pjsub go4.sh
```

S1-1：局所ベクトル読み込み，ノルム計算
計算結果

予め求めておいた答え

```
a1.* 1.62088247569032590000E+03
a2.* 1.22218492872396360000E+03

$> ifort -O3 dot-a1.f
$> pjsub go1.sh

$> ifort -O3 dot-a2.f
$> pjsub go1.sh
```

計算結果

```
a1.* 1.62088247569032590000E+03
a2.* 1.22218492872396360000E+03
```

go1.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst
```

S1-1：局所ベクトル読み込み，ノルム計算
SENDBUFとRECVBUFを同じにしたら・・・

正
```
MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM,
                   MPI_COMM_WORLD)
```

誤
```
MPI_Allreduce(&sum0, &sum0, 1, MPI_DOUBLE, MPI_SUM,
                   MPI_COMM_WORLD)
```

S1-1：局所ベクトル読み込み，ノルム計算
SENDBUFとRECVBUFを同じにしたら・・・

正
```
MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD)
```

誤
```
MPI_Allreduce(&sum0, &sum0, 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD)
```

正
```
MPI_Allreduce(&sumK[1], &sumK[2], 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD)
```

これバッファが重なっていないのでOK

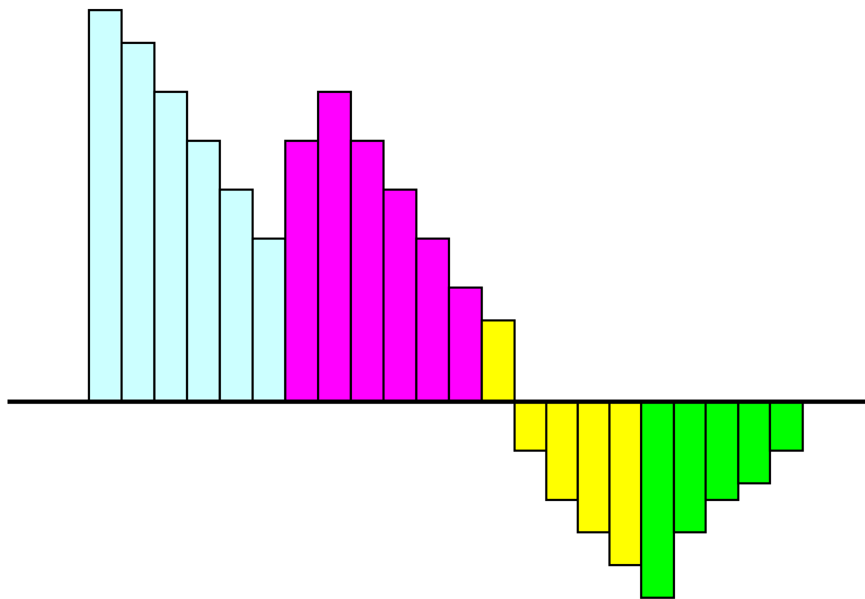
S1-3：台形則による積分

- 下記の数値積分の結果を台形公式によって求めるプログラムを作成する。MPI_REDUCE，MPI_BCASTを使用して並列化を実施し，プロセッサ数を変化させた場合の計算時間を測定する。

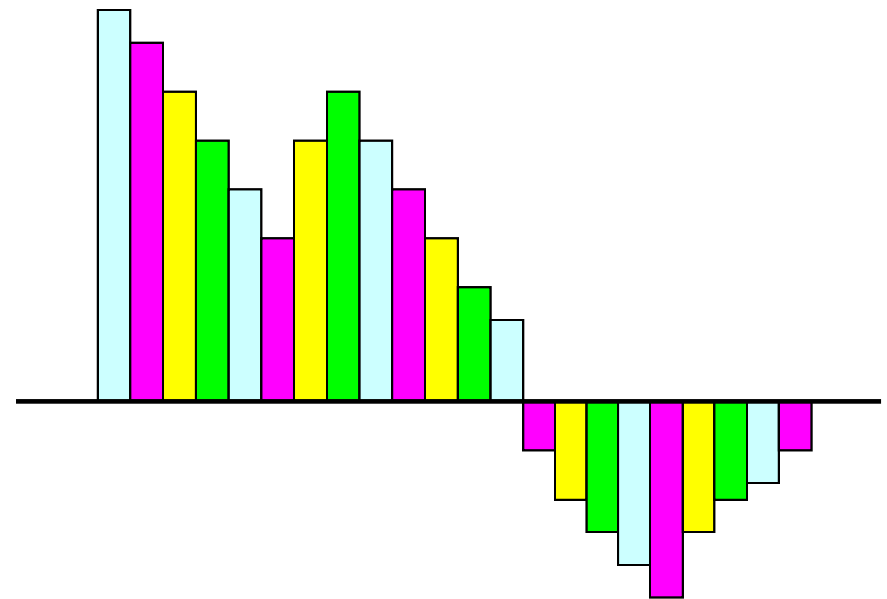
$$\int_0^1 \frac{4}{1+x^2}\,dx$$

S1-3：台形則による積分
プロセッサへの配分の手法

タイプA



タイプB



$$\frac{1}{2}\Delta x\left(f_1 + f_{N+1} + \sum_{i=2}^{N} 2 f_i\right)$$

を使うとすると必然的に「タイプA」となるが・・・

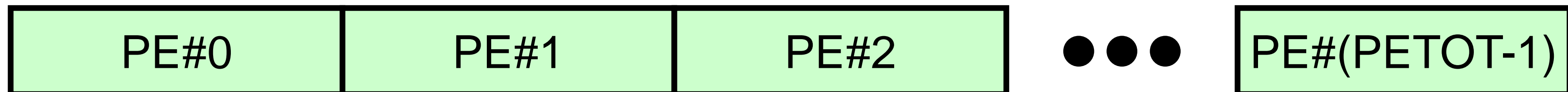
S1-3

S1-3：台形則による計算

TYPE-A（1/2）：s1-3a.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "mpi.h"

int main(int argc, char **argv){
        int i;
        double TimeStart, TimeEnd, sum0, sum, dx;
        int PeTot, MyRank, n, int *index;
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        index = calloc(PeTot+1, sizeof(int));
        fp = fopen("input.dat", "r");
        fscanf(fp, "%d", &n);
        fclose(fp);
        if(MyRank==0) printf("%s%8d¥n", "N=", n);
        dx = 1.0/n;

        for(i=0;i<=PeTot;i++){
                index[i] = ((long long)i * n)/PeTot;}
```

“input.dat”で分割数Nを指定
中身を書き出して見よう:n

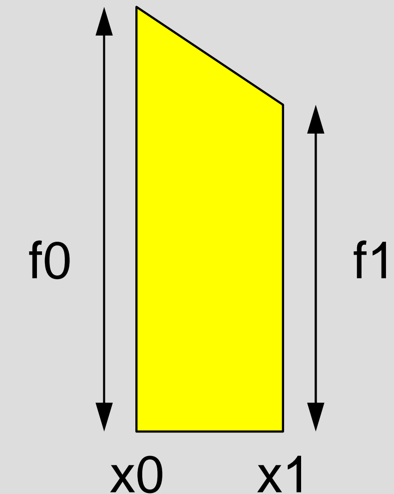


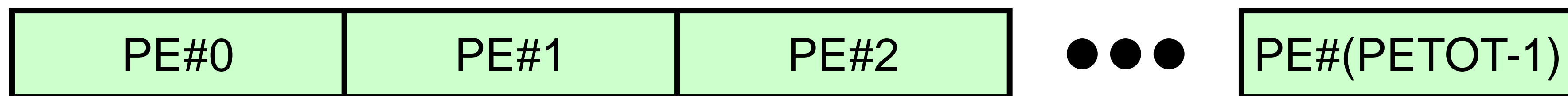
index[0] index[1] index[2] index[3] index[PETOT-1] index[PeTot]=N

S1-3：台形則による計算

TYPE-A (2/2)：s1-3a.c

```
        TimeS = MPI_Wtime();
        sum0 = 0.0;
        for(i=index[MyRank]; i<index[MyRank+1]; i++)

        {
                double x0, x1, f0, f1;
                x0 = (double)i * dx;
                x1 = (double)(i+1) * dx;
                f0  = 4.0/(1.0+x0*x0);
                f1  = 4.0/(1.0+x1*x1);
                sum0 += 0.5 * (f0 + f1) * dx;
        }

        MPI_Reduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        TimeE = MPI_Wtime();

        if(!MyRank) printf("%24.16f%24.16f%24.16f¥n", sum, 4.0*atan(1.0), TimeE - TimeS);

        MPI_Finalize();
        return 0;
}
```



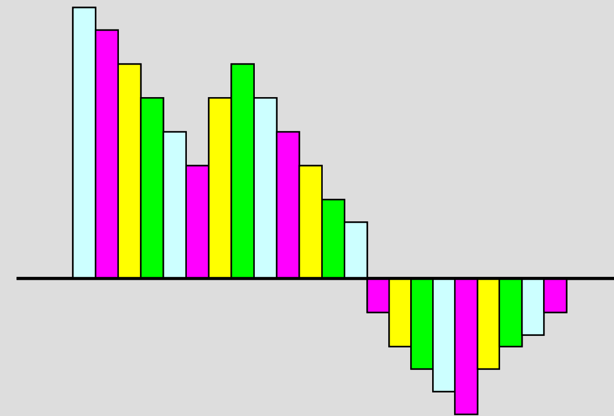


index[0] index[1] index[2] index[3] index[PETOT-1] index[PeTot] =N

S1-3：台形則による計算

TYPE-B ：s1-3b.c

```
        TimeS = MPI_Wtime();
        sum0 = 0.0;
        for(i=MyRank; i<n; i+=PeTot)

        {
                double x0, x1, f0, f1;
                x0 = (double)i * dx;
                x1 = (double)(i+1) * dx;
                f0  = 4.0/(1.0+x0*x0);
                f1  = 4.0/(1.0+x1*x1);
                sum0 += 0.5 * (f0 + f1) * dx;
        }

        MPI_Reduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        TimeE = MPI_Wtime();

        if(!MyRank) printf("%24.16f%24.16f%24.16f¥n", sum, 4.0*atan(1.0), TimeE-TimeS);

        MPI_Finalize();
        return 0;
}
```



コンパイル・実行（課題S1-3）

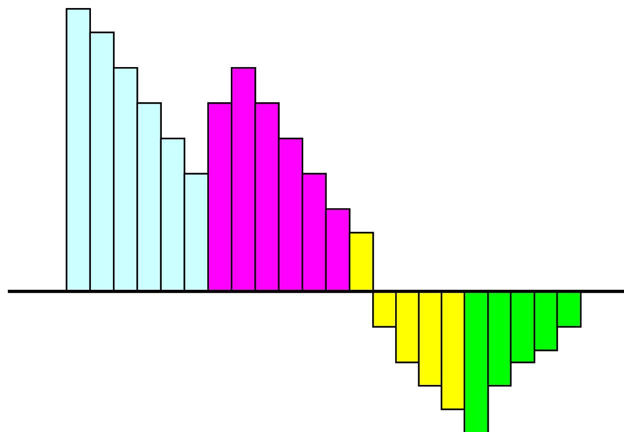
```
$ mpiifort -align array64byte -O3 -axCORE-AVX512 s1-3a.f -o testa
$ mpiifort -align array64byte -O3 -axCORE-AVX512 s1-3b.f -o testb

(modify "go.sh")
$ pjsub go.sh
```

FORTRAN

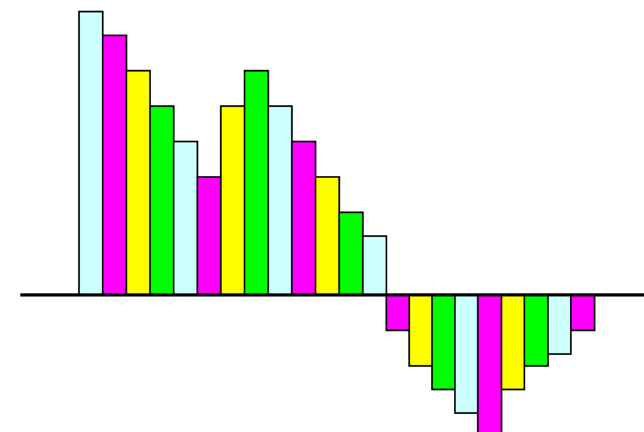
```
$ mpiicc -align -O3 -axCORE-AVX512 s1-3a.c -o testa
$ mpiicc -align -O3 -axCORE-AVX512 s1-3b.c -o testb

(modify "go.sh")
$ pjsub go.sh
```

C

Type-A



Type-B



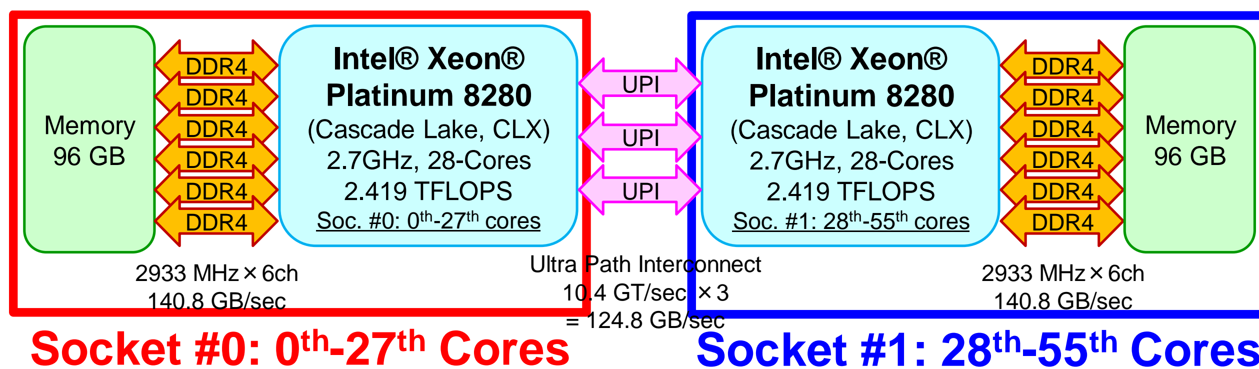
go.sh: 8-nodes, 384-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb
```

384/8= 48 cores/node

48-cores are randomly selected from 56-cores on the node

24-cores on each socket are assigned.
A little bit more stable

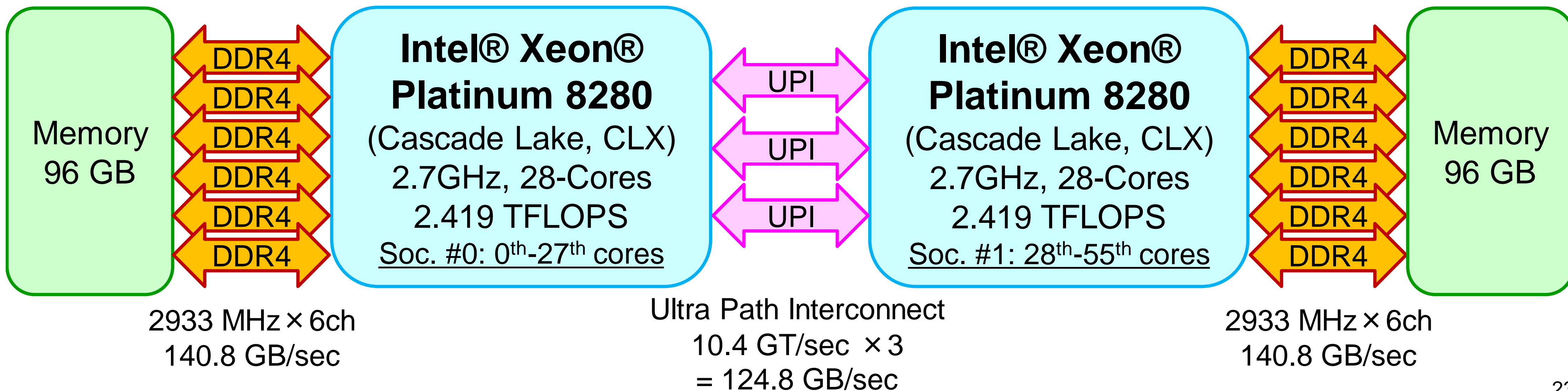


Socket #0: 0th-27th Cores

Socket #1: 28th-55th Cores

Process Number

```
#PJM -L node=1;#PJM --mpi proc= 1    1-node, 1-proc, 1-proc/n
#PJM -L node=1;#PJM --mpi proc= 4    1-node, 4-proc, 4-proc/n
#PJM -L node=1;#PJM --mpi proc=16    1-node,16-proc,16-proc/n
#PJM -L node=1;#PJM --mpi proc=28    1-node,28-proc,28-proc/n
#PJM -L node=1;#PJM --mpi proc=56    1-node,56-proc,56-proc/n

#PJM -L node=4;#PJM --mpi proc=128   4-node,128-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=256   8-node,256-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=448   8-node,448-proc,56-proc/n
```



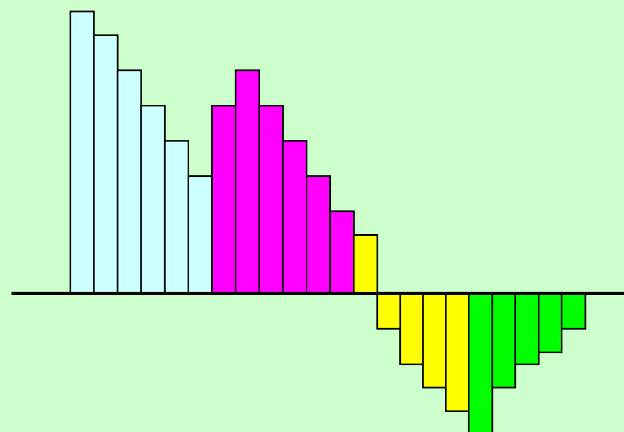
S1-3：OBCX上での性能評価

- 1コア当たり使用時の計算時間を元にする
- 5回実行した最速ケースを採択
- Type1/2-A/B (次頁)
  - Type1：コアは56コアからランダムに選択される
  - Type2：使用コアは指定，両ソケットでバランス
- Strong Scaling
  - 全体の問題規模は固定
  - N倍のコア数⇒1/Nの計算時間
- Weak Scaling
  - コア当たり問題規模固定
  - N倍のコア数，計算時間一定（普通はそうはならない）

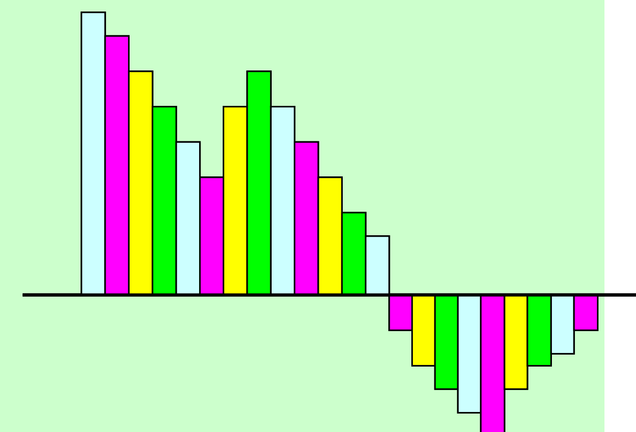
go.sh: 8-nodes, 384-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb
```

Type-a

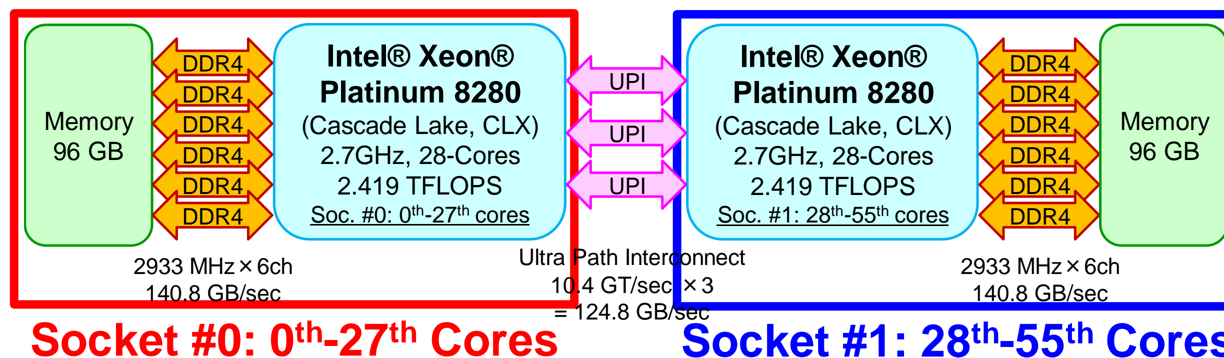


Type-b

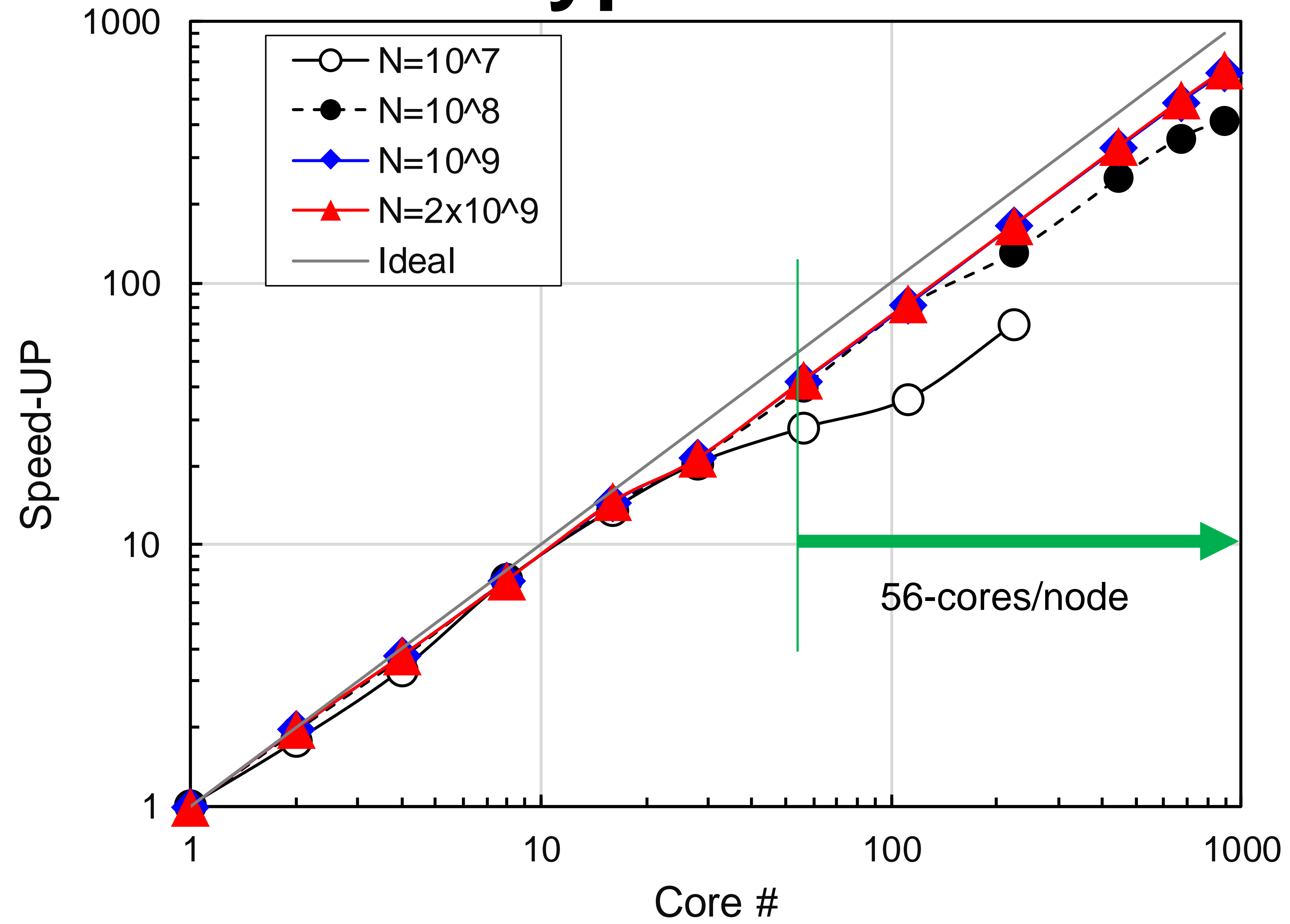


Type 1-a
Type 1-b

Type 2-a
Type 2-b



Strong Scaling up: 16-nodes, 896-cores Type-2-a

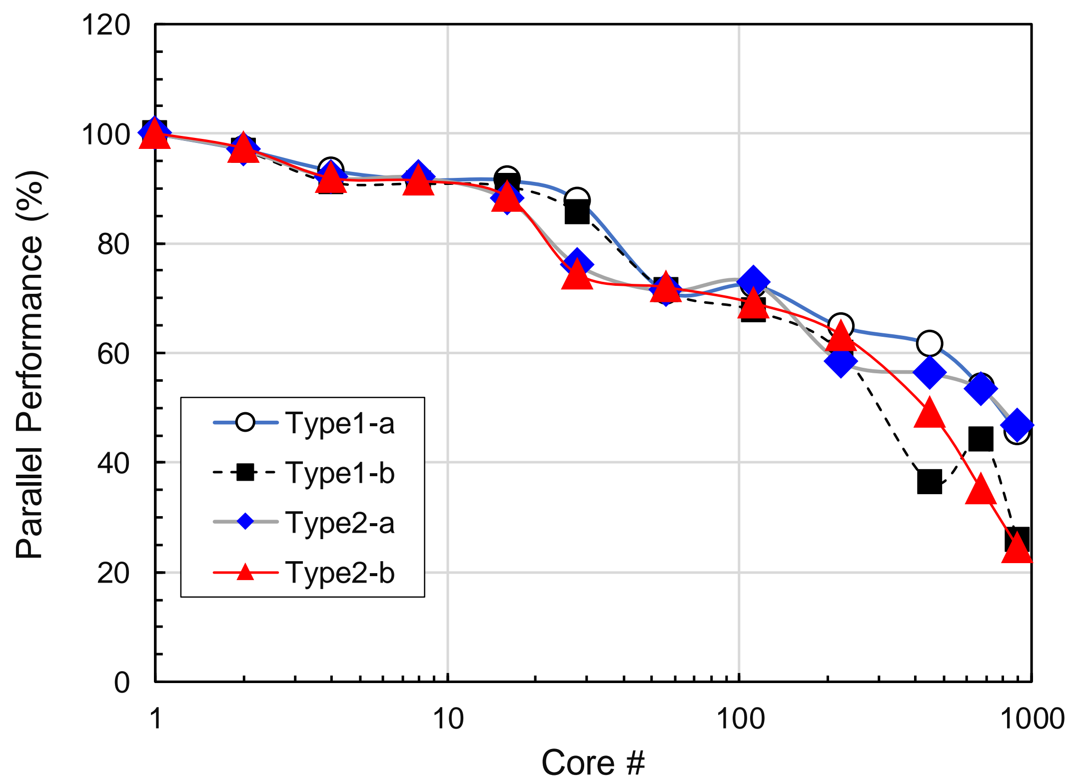


Parallel Performance

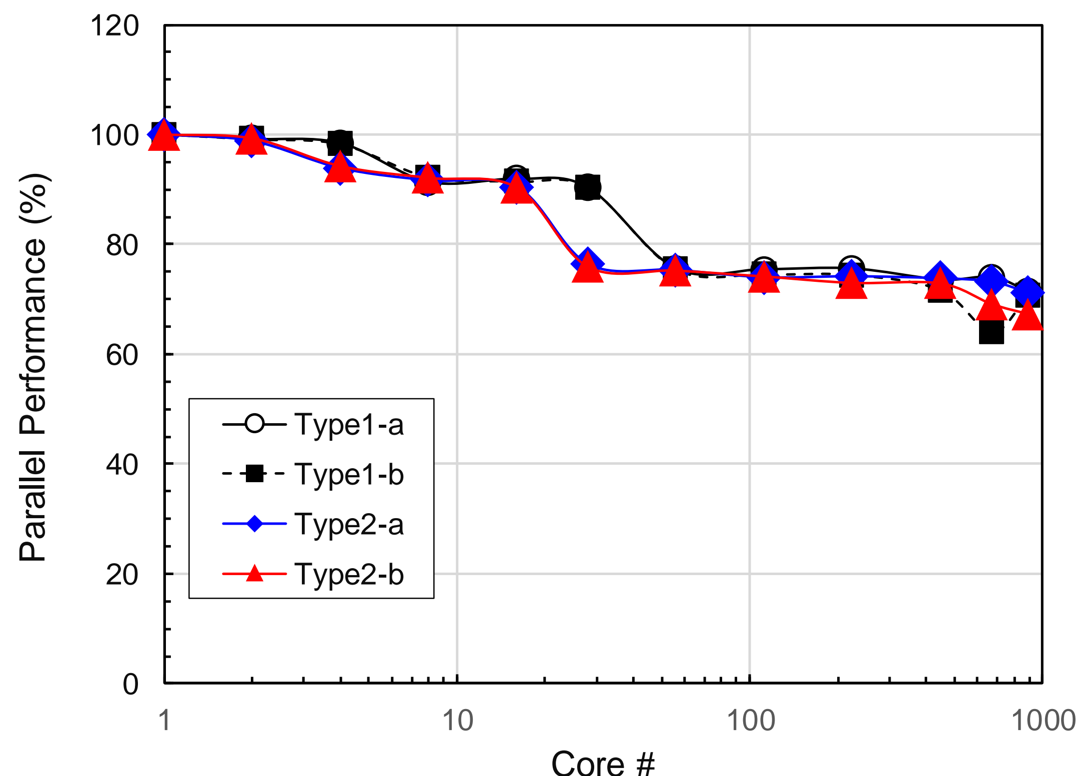
Performance of Type 1-a with 1-core= 100%

Nが大きいとコア数が増加する程性能は高い
コア数が多いとType-AがType-Bより良い
Type-1とType-2の差はほとんど無い

N= 10^8



N= 10^9



Parallel Performance 並列化効率

Number of PE's	Computation Time (sec)	Parallel Performance(%) (based on performance with 1PE)
1	100	-
100	1.00	100%
100	1.50	66.7% = (1.00/1.50)x100

At 28-cores

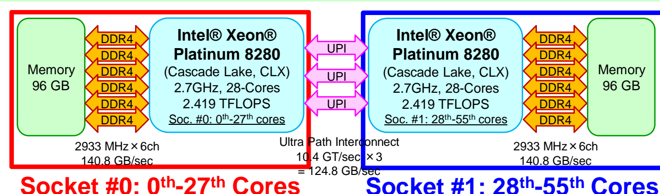
- Type1：○■：28-cores are randomly selected from 2 Sockets (e.g. 14-cores on 1st Soc, 14-cores on 2nd Soc)
- Type2：◆▲：28-cores are on the 1st Socket
- Type1の方がメモリ，キャッシュは有効に利用されている

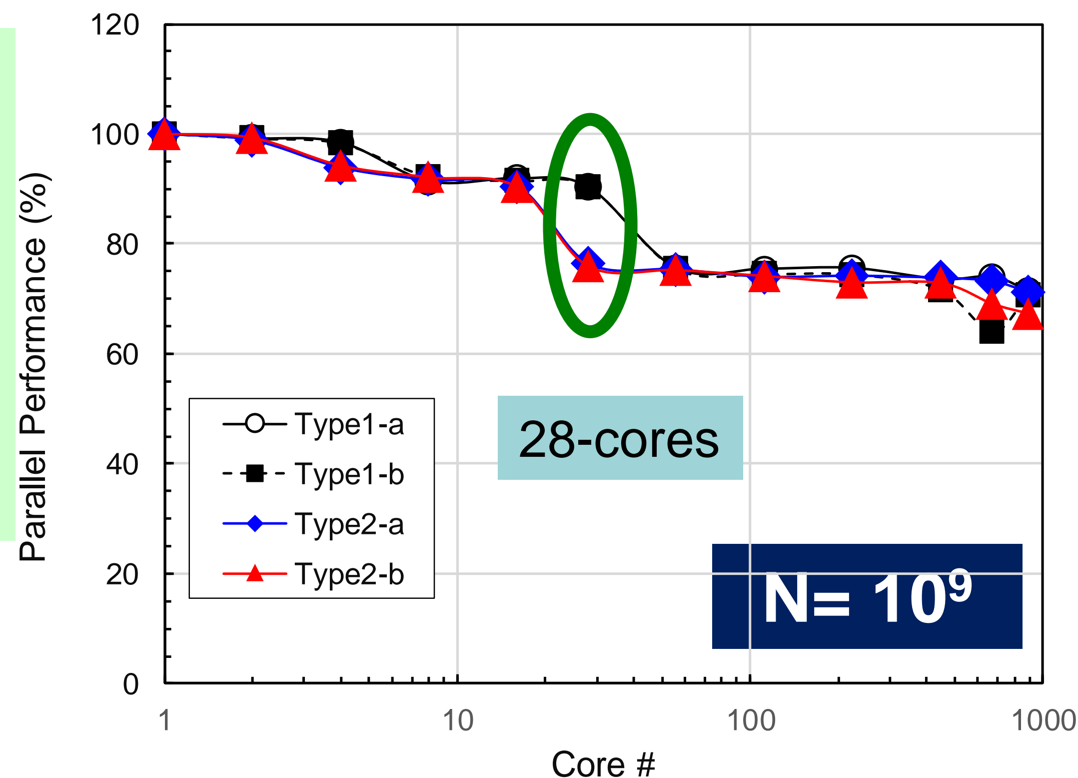
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture7
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt37
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testb
```

384/8= 48 cores/node

48-cores are randomly selected from 56-cores on the node

24-cores on each socket are assigned. A little bit more stable

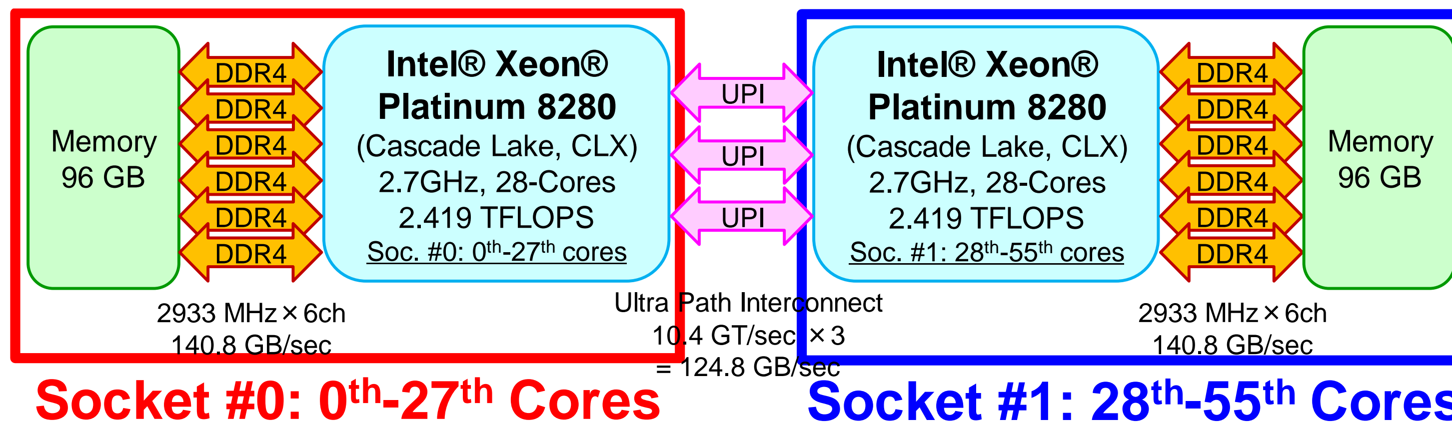




With/Without Numactl: b48.sh

Not so different

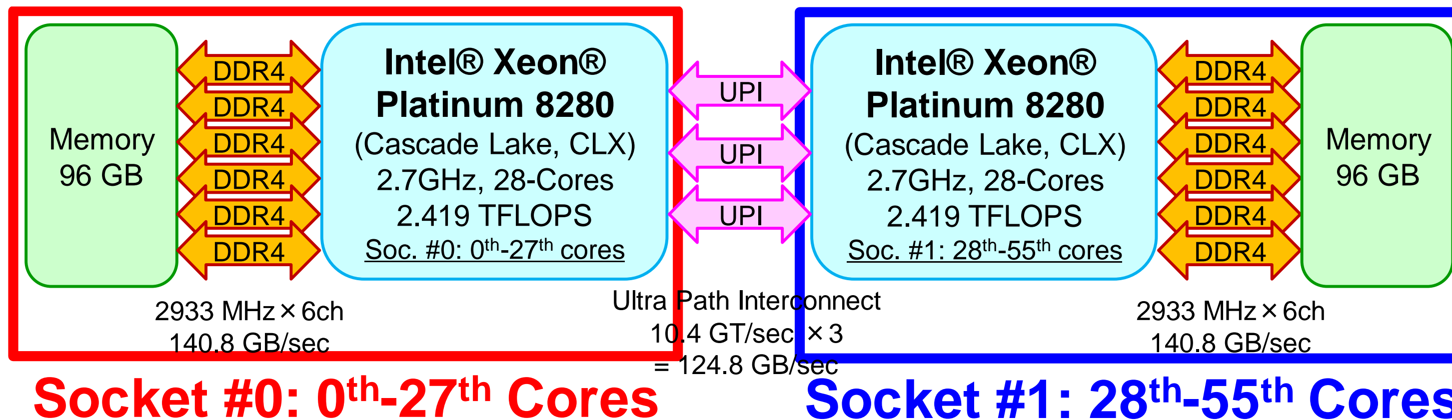
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=384                          384÷8= 48-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
```



With/Without Numactl: b56.sh

Not so different

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=448                    448÷8= 56-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-55 (not needded)
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./testa
mpiexec.hydra -n ${PJM_MPI_PROC} ./testa
```

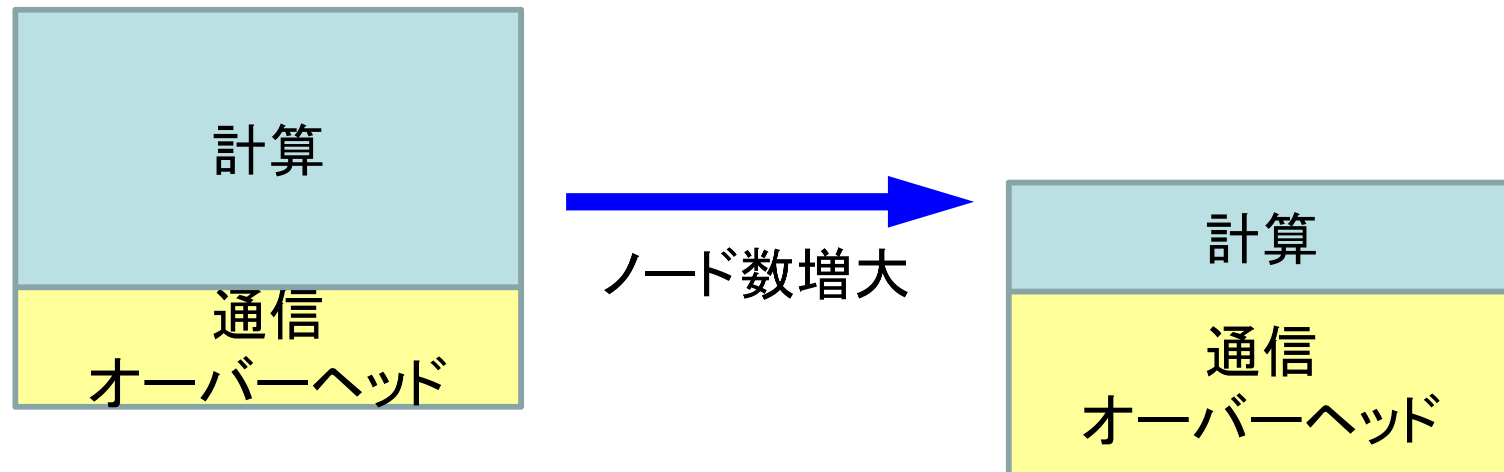


理想値からのずれ

- MPI通信そのものに要する時間
  - データを送付している時間
  - ノード間においては通信バンド幅によって決まる
    - Gigabit Ethernetでは 1Gbit/sec.（理想値）
  - 通信時間は送受信バッファのサイズに比例
- MPIの立ち上がり時間
  - latency
  - 送受信バッファのサイズによらない
    - 呼び出し回数依存，プロセス数が増加すると増加する傾向
  - 通常，数～数十μsecのオーダー
- MPIの同期のための時間
  - プロセス数が増加すると増加する傾向

理想値からのずれ（続き）

- 計算時間が小さい場合（Nが小さい場合）はこれらの効果を無視できない。
  - 特に，送信メッセージ数が小さい場合は，「Latency」が効く。
  - 粒度（granularity）：プロセス当たり問題サイズ

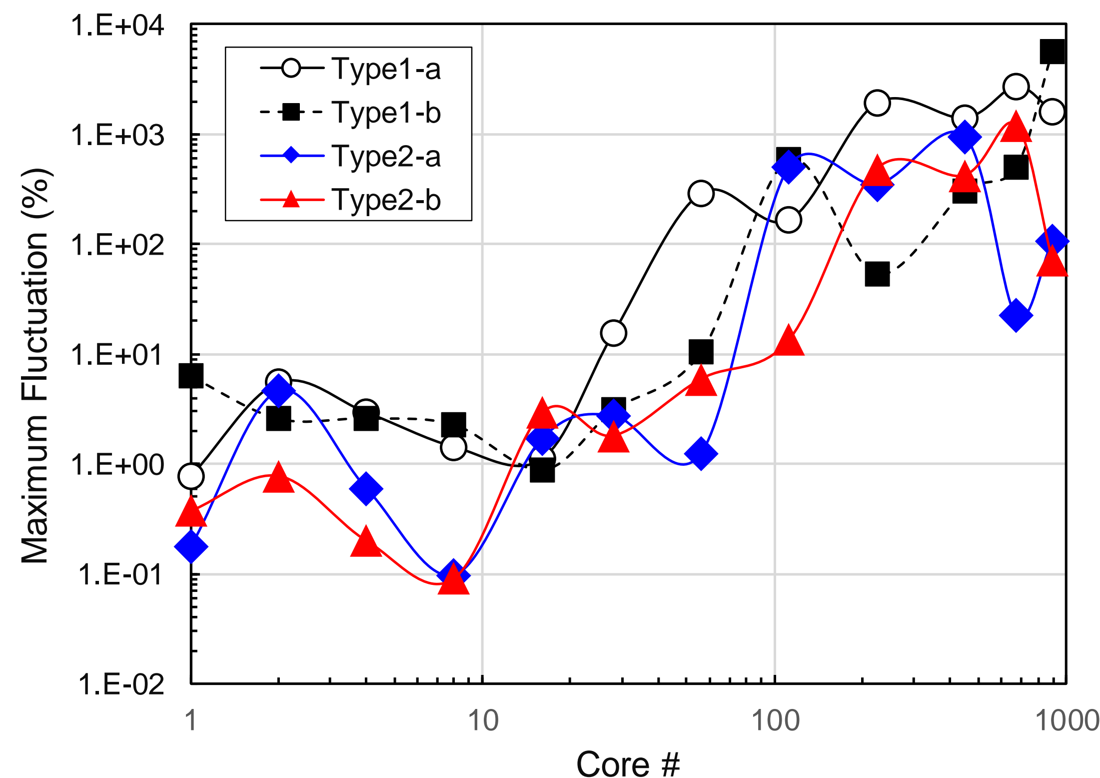


最大変動率（5回）: $100*(T_{max}-T_{min})/T_{min}$

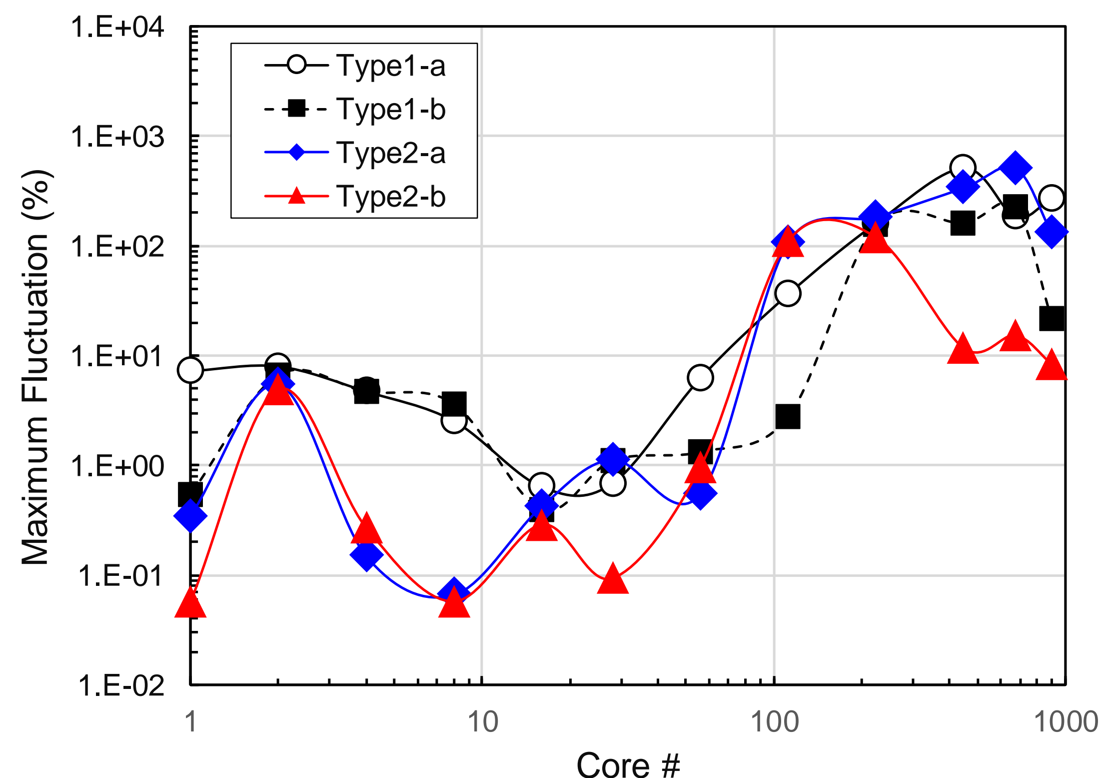
問題規模が大きい程，コア数が増えた場合の変動率は少ない

Type-2/Type-BはType-1/Type-Aより良いが挙動はランダム

N= 10^8



N= 10^9



Parallel Performance

Number of PE's	Computation Time (sec)	Parallel Performance(%) (based on performance with 1PE)
1	100	-
100	1.00	100%
100	1.50	66.7% = (1.00/1.50)x100